Contents

[Section 1 2](#_Toc151557065)

[Section 2 4](#_Toc151557066)

[Introduction to REST API and best practices 4](#_Toc151557067)

[Creating Spring Boot project 5](#_Toc151557068)

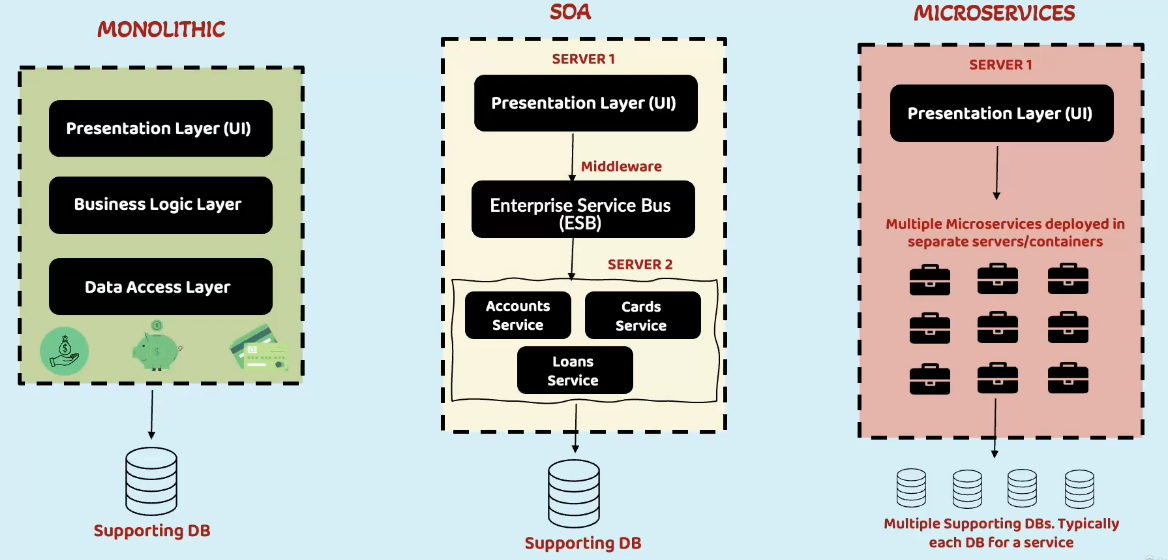
[Configuring H2 DB and yaml properties 6](#_Toc151557069)

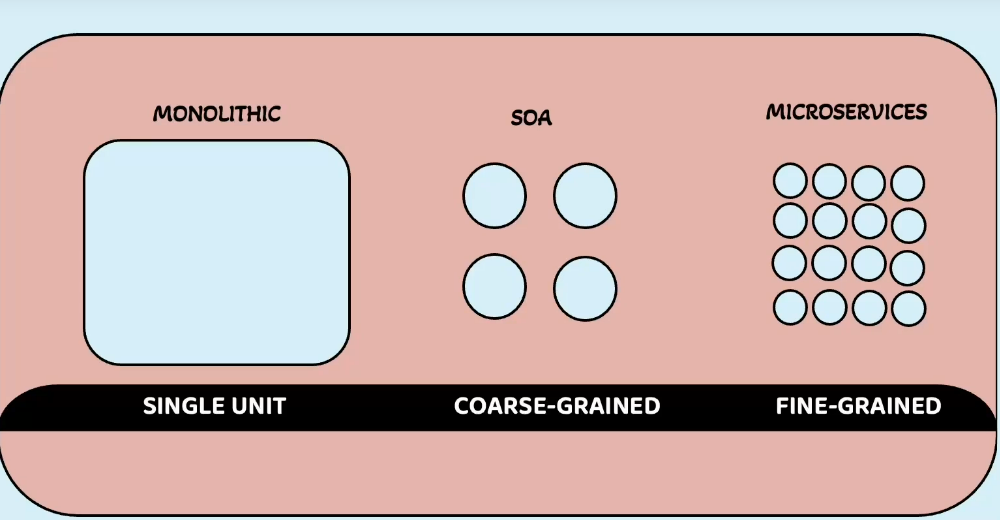
[Introduction to DTO (data transfer objects) 11](#_Toc151557070)

[Creating DTO’s inside accounts microservice 11](#_Toc151557071)

[Create API inside accounts microservice – Part 1 13](#_Toc151557072)

[Create API inside accounts microservice – Part 2 16](#_Toc151557073)

Section 1  
Difference between Monolithic, SOA and Microservices  


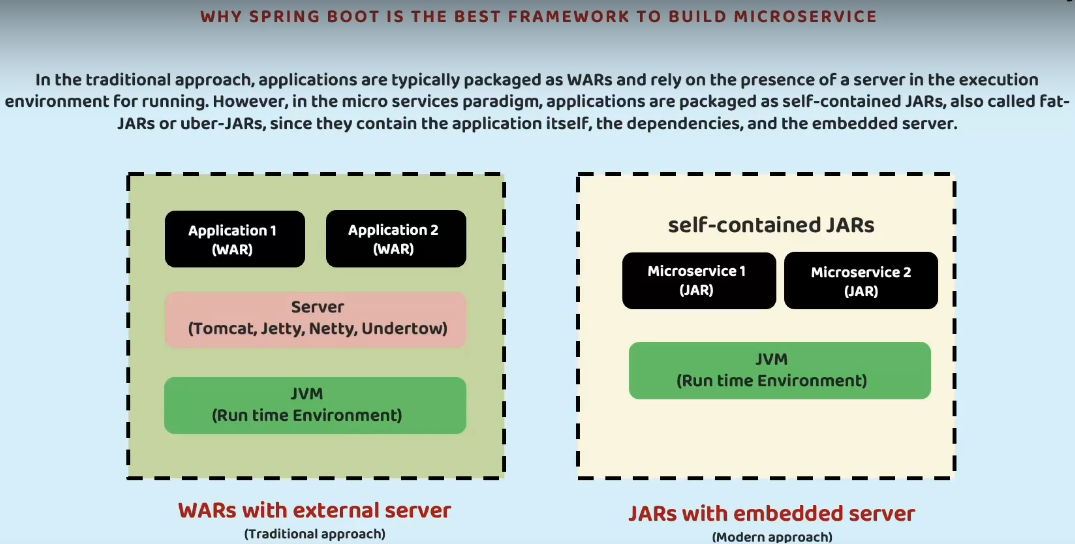
**Monolithic:**All applications will be deployed in a single server, supported by a single database.  
The web application and the business logic is tightly coupled.  
  
**SOA:**We separate the UI and the backend logic in different servers, but again supported by a single database.   
We add a middle ware extra component (Enterprise Service Bus - ESB) which is difficult to maintain.  
There is some level of separation but not based on business domain.  
  
**Microservice:**Here we create separate deployable services based on business domain and each has its own database. All these microservices and databases can be a mix of technology.   
So, each service has its own separate development, code base and deployment lifecycle.  
  


|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **Monolithic** | **SOA** | **Microservices** |
| Parallel development by teams on features which are component specific | NO | MIDDLE (separation between UI and backend teams) | YES |
| Agility (Enhancing the application or component with new language / framework) | NO | MIDDLE (separation between UI and backend teams) | YES |
| Scalability | NO - Not possible because you are going to deploy all your application in one single jumbo server, so to scale you bring in one more jumbo server and need to take care of load balancing – manual setup | MIDDLE – Difficult to scale you, as are going to deploy all your application in one single jumbo server. | YES – Scalability is easy, because of products like docker and Kubernetes. |
| Usability | NO | MIDDLE | YES – Any new feature specific for domain component can be deployed without any interference to any other component. |
| Complexity and Operational Overhead | YES – Here there is only one server, you have to make sure just that one server is running without any issues | MIDDLE – Here you need to take care of UI server, middle ware server and backend server. | NO |
| Security and Performance | YES – Here there use to be method calls | MIDDLE | NO – Here the communication between components/service are REST services, so request goes over the network. So there may be some network latency. |

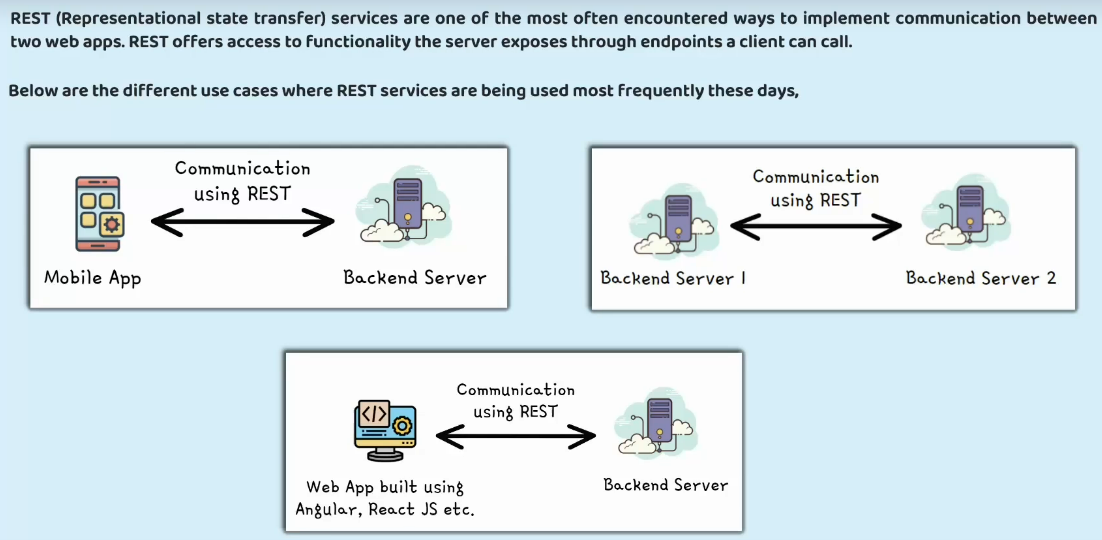
Definition of Microservices  


Section 2  
How to build Microservices  

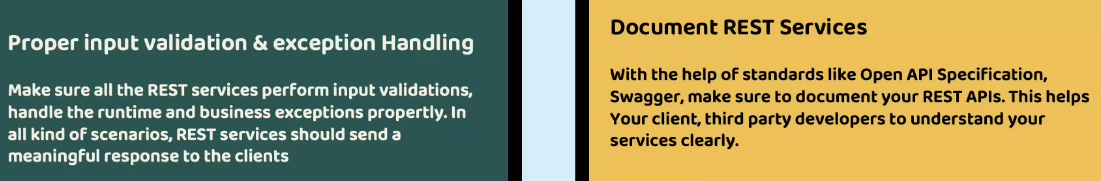


## Introduction to REST API and best practices







Creating Spring Boot project  
<parent>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-parent</artifactId>

        <version>3.1.5</version>

</parent>

<groupId>com.eazybytes</groupId>

<artifactId>accounts</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-actuator</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-validation</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-web</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-devtools</artifactId>

            <scope>runtime</scope>

            <optional>true</optional>

        </dependency>

        <dependency>

            <groupId>com.h2database</groupId>

            <artifactId>h2</artifactId>

            <scope>runtime</scope>

        </dependency>

        <dependency>

            <groupId>org.projectlombok</groupId>

            <artifactId>lombok</artifactId>

            <optional>true</optional>

        </dependency>

    </dependencies>

import org.springframework.boot.autoconfigure.SpringBootApplication;  
With this annotation we are telling spring boot framework to perform auto configurations and to scan all the beans inside the spring boot application.  
So **@SpringBootApplication** is a combination of all these annotations   
@Inherited  
@SpringBootConfiguration  
@EnableAutoConfiguration  
@ComponentScan  
  
Creating Hello World REST API using @RestController   
Whenever we are putting an **@RestController** annotation on top of a class we are instructing the framework, that I am going to write method in this class related to http methods, so expose all these methods as a REST API to the outside world.  
  
package com.eazybytes.accounts.controller;

@RestController

public class AccountsController {

@GetMapping("sayHello")

public String sayHello(){

return "Say Hello";

}

}

Configuring H2 DB and yaml properties  
**accounts\src\main\resources\application.properties rename to application.yml**  
spring.datasource.url.=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

server:

port: 8080

spring:

datasource:

url: jdbc:h2:mem:testdb

driver-class-name: org.h2.Driver

username: sa

password:

h2:

console:

enabled: true

jpa:

database-platform: org.hibernate.dialect.H2Dialect

hibernate:

ddl-auto: update

show-sql: true

**accounts\src\main\resources\schema.sql**CREATE TABLE IF NOT EXISTS `customer` (

`customer\_id` int AUTO\_INCREMENT PRIMARY KEY,

`name` varchar(100) NOT NULL,

`email` varchar(100) NOT NULL,

`mobile\_number` varchar(20) NOT NULL,

`created\_at` date NOT NULL,

`created\_by` varchar(20) NOT NULL,

`updated\_at` date DEFAULT NULL,

`updated\_by` varchar(20) DEFAULT NULL

);

CREATE TABLE IF NOT EXISTS `accounts` (

`customer\_id` int NOT NULL,

`account\_number` int AUTO\_INCREMENT PRIMARY KEY,

`account\_type` varchar(100) NOT NULL,

`branch\_address` varchar(200) NOT NULL,

`created\_at` date NOT NULL,

`created\_by` varchar(20) NOT NULL,

`updated\_at` date DEFAULT NULL,

`updated\_by` varchar(20) DEFAULT NULL

);

Start the spring application  
<http://localhost:8080/h2-console/>

You will see that the 2 tables are created ‘**customer’** and ‘**accounts’**.  
  
Writing Spring Data JPA Entities and repositories to interact with DB tables

For this we have already added the library spring-boot-starter-data-jpa in which we have many interfaces and classes which we can use to interact with the database.  
  
Before using this framework, we need to create a POJO or entity classes which represent the tables in our database.  
  
CREATE TABLE IF NOT EXISTS `customer` (

`customer\_id` int AUTO\_INCREMENT PRIMARY KEY,

`name` varchar(100) NOT NULL,

`email` varchar(100) NOT NULL,

`mobile\_number` varchar(20) NOT NULL,

`created\_at` date NOT NULL,

`created\_by` varchar(20) NOT NULL,

`updated\_at` date DEFAULT NULL,

`updated\_by` varchar(20) DEFAULT NULL

);

CREATE TABLE IF NOT EXISTS `accounts` (

`customer\_id` int NOT NULL,

`account\_number` int AUTO\_INCREMENT PRIMARY KEY,

`account\_type` varchar(100) NOT NULL,

`branch\_address` varchar(200) NOT NULL,

`created\_at` date NOT NULL,

`created\_by` varchar(20) NOT NULL,

`updated\_at` date DEFAULT NULL,

`updated\_by` varchar(20) DEFAULT NULL

);

There are 4 columns which are common to each of these tables. These are called metadata columns. So, these should have a separate super class **BaseEntity**.

`created\_at` date NOT NULL, so the field name should be createdAt  
`created\_by` varchar(20) NOT NULL, so the field name should be createdBy  
So no need to mention @Column(name=”created\_at”) for fields

By doing this we do not have the need to tell the framework which column this field maps to.  
  
**@MappedSuperclass** this signify that this entity class will act the super class for all you entity classes.  
  
**@Column**(**updatable** = false) when the row is being updated we don’t want this column to be considered by spring data JPA for updating, this fields will not be updated whenever we update the record. It will be populated only once when the record is inserted or created.  
  
**@Column(insertable** = false) this tells do not populate this column whenever a new record is inserted or created.  
  
package com.eazybytes.accounts.entity;

import jakarta.persistence.MappedSuperclass;

@MappedSuperclass

@Getter @Setter @ToString

public class BaseEntity {

@Column(updatable = false)

private LocalDateTime createdAt;

@Column(updatable = false)

private String createdBy;

@Column(insertable = false)

private LocalDateTime updatedAt;

@Column(insertable = false)

private String updatedBy;

}

**@Entity** this is telling the JPA framework please treat this class as entity or POJO representation of the table ‘customer’. Table name should match with the class name Customer.

If it does not match you can use annotation @Table(name=”customer”) on top of the Entity class.  
  
**@AllArgsConstructor @NoArgsConstructor** these are required when we try to create the object of this class.  
  
**@Id** is used to signify the primary key of the table, but are we going to provide the value to the primary key manually   
or  
we want the spring data framework to generate the value by itself when we try to insert a new record inside the table.

Here in this case, it is good to give the responsibility to the spring data framework. For this   
**@GeneratedValue(strategy = GenerationType.AUTO,generator = "native")**

@GenericGenerator(name = "native", strategy = "native")  
This tells the spring data framework generate the value automatically, With “**native**” strategy we are telling spring data framework, whatever database that I am using please try to generate the sequence number or the primary key value based upon the native style of my database.

package com.eazybytes.accounts.entity;

@Entity

@Getter @Setter @ToString @AllArgsConstructor @NoArgsConstructor

public class Customer extends BaseEntity{

@Id

@GeneratedValue(strategy = GenerationType.AUTO,generator = "native")

@GenericGenerator(name = "native", strategy = "native")

@Column(name = "customer\_id")

private Long customerId;

private String name;

private String email;

@Column(name = "mobile\_number")

private String mobileNumber;

}

For accountNumber which is the primary key we will have our own logic to generate the value. Because we don’t account number to start with 1, 2, etc... account numbers are usually 10 digit numbers.  
  
package com.eazybytes.accounts.entity;

@Entity

@Getter @Setter @AllArgsConstructor @NoArgsConstructor

public class Accounts extends BaseEntity{

@Id

@Column(name = "account\_number")

private Long accountNumber;

@Column(name = "account\_type")

private String accountType;

@Column(name = "branch\_address")

private String branchAddress;

}

Now we should also create the repository classes.  
Create package com.eazybytes.accounts.repository  
**@Repository** when we mention this annotation, spring data JPA framework will create the bean implementation (runtime code having lots of methods for various sorts of CRUD operations) of this interface in the background based on the configurations that we provided in application.yml and this interface, extends JpaRepository<Customer, Long> so this represents which is the entity class which is going to be handled by this repository class and what is the datatype of your primary key.

JpaRepository internally extends various interfaces ListCrudRepository, ListPagingAndSortingRepository, QueryByExampleExecutor   
  
package com.eazybytes.accounts.repository;

import org.springframework.data.jpa.repository.JpaRepository;

@Repository

public interface CustomerRepository extends JpaRepository<Customer, Long>{

}

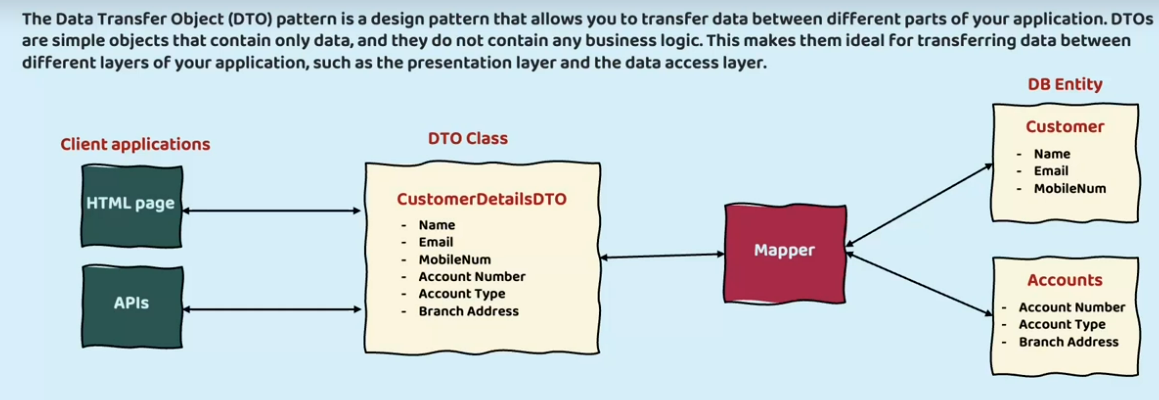
package com.eazybytes.accounts.repository;

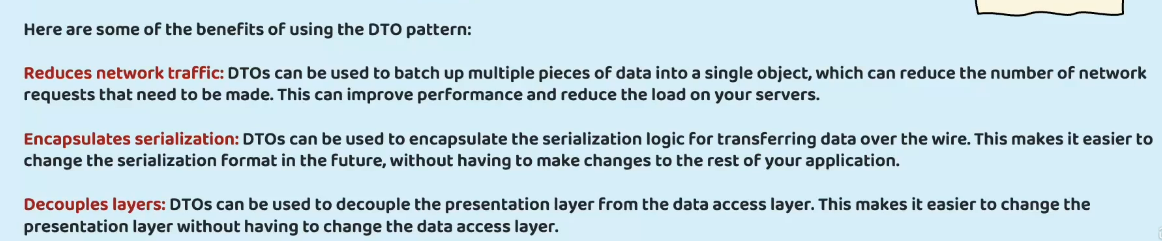
import org.springframework.data.jpa.repository.JpaRepository;

@Repository

public interface AccountsRepository extends JpaRepository<Accounts, String> {

}

Introduction to DTO (data transfer objects)  
So entity classes are used to create, update, delete accounts and customer detail or records inside a database.   
But what if a client application wants you accounts and customer details together in a single request. So, we cannot send 2 different objects inside a response, we can send only 1 single object inside a response.  
These entity classes are related to the database layer,   
  




Creating DTO’s inside accounts microservice  
create package com.eazybytes.accounts.dto  
  
**@Data** this not only creates the getters and setters but also the hashcode and equals methods. But this does not create a constructor which accepts all fields as arguments.   
  
package com.eazybytes.accounts.dto;

@Data

public class AccountsDto {

private Long accountNumber;

private String accountType;

private String branchAddress;

}

package com.eazybytes.accounts.dto;

@Data

public class CustomerDto {

private String name;

private String email;

private String mobileNumber;

}

package com.eazybytes.accounts.dto;

@Data @AllArgsConstructor

public class ResponseDto {

private String statusCode;

private String statusMsg;

}

package com.eazybytes.accounts.dto;

@Data @AllArgsConstructor

public class ErrorResponseDto {

private String apiPath;

private HttpStatus errorCode;

private String errorMsg;

private LocalDateTime errorTime;

}

## Create API inside accounts microservice – Part 1

Create a constants class as below  
package com.eazybytes.accounts.constants;

public final class AccountsConstants {

private AccountsConstants() {

// restrict instantiation

}

public static final String SAVINGS = "Savings";

public static final String ADDRESS = "123 Main Street, New York";

public static final String STATUS\_201 = "201";

public static final String MESSAGE\_201 = "Account created successfully";

public static final String STATUS\_200 = "200";

public static final String MESSAGE\_200 = "Request processed successfully";

public static final String STATUS\_417 = "417";

public static final String MESSAGE\_417\_UPDATE= "Update operation failed. Please try again or contact Dev team";

public static final String MESSAGE\_417\_DELETE= "Delete operation failed. Please try again or contact Dev team";

// public static final String STATUS\_500 = "500";

// public static final String MESSAGE\_500 = "An error occurred. Please try again or contact Dev team";

}

It is always recommended to maintain a prefix API path which is common to the rest API present inside your controller class.

@RestController

**@RequestMapping(path = "/api", produces = MediaType.APPLICATION\_JSON\_VALUE)**

public class AccountsController {  
 ….

}  
  
**@RequestBody** = Whatever the clients are sending in the request body the same will be mapped into the POJO class that we mention.  
  
package com.eazybytes.accounts.controller;

@RestController

@RequestMapping(path = "/api", produces = MediaType.APPLICATION\_JSON\_VALUE)

public class AccountsController {

@PostMapping("/create")

public ResponseEntity<ResponseDto> createAccount(@RequestBody CustomerDto customerDto)

{

return ResponseEntity

.status(HttpStatus.CREATED)

.body(new ResponseDto(AccountsConstants.MESSAGE\_201, AccountsConstants.MESSAGE\_201));

}

}

The business logic should always be in a service layer,   
the Controller layer should only   
a. accept the request and send the response.

b. perform any validations

Create package com.eazybytes.accounts.service  
package com.eazybytes.accounts.service;

public interface IAccountsService {

void createAccount(CustomerDto customerDto);

}  
  
Create package com.eazybytes.accounts.service.impl  
**@Service** – To indicate the spring framework that this class is going to act as a service layer so please create the bean and once the bean is created by the spring boot framework , we can auto wire that bean in the controller layer.  
  
package com.eazybytes.accounts.service.impl;

@Service

public class AccountsServiceImpl implements IAccountsService {

@Override

public void createAccount(CustomerDto customerDto) {

throw new UnsupportedOperationException("Unimplemented method 'createAccount'");

}

}

Before we write the logic to create the account, auto wire the repository classes.  
@Service

@AllArgsConstructor

public class AccountsServiceImpl implements IAccountsService {  
….  
}  
***Because of @AllArgsConstructor it will create an all args constructor , so when we have only single constructor inside your class that is accepting parameters, you don’t need to auto wire these repository classes with the help of @Autowire.  
Spring framework will automatically do the auto wiring, since there is single constructor.***  
Now when we save the details of the customer we need to sends the object of Customer entity object and not the DTO class object. So, there should be some mapping logic to convert the DTO to entity object.  
  
package com.eazybytes.accounts.service.impl;

@Service

@AllArgsConstructor

public class AccountsServiceImpl implements IAccountsService {

private AccountsRepository accountsRepository;

private CustomerRepository customerRepository;

@Override

public void createAccount(CustomerDto customerDto) {

throw new UnsupportedOperationException("Unimplemented method 'createAccount'");

}

}

Create package com.eazybytes.accounts.mapper  
These mapper classes will take the responsibility of converting the entity to DTO and DTO to entity.  
  
package com.eazybytes.accounts.mapper;

public class AccountsMapper {

public static AccountsDto mapToAccountDto(Accounts accounts, AccountsDto accountsDto){

accountsDto.setAccountNumber(accounts.getAccountNumber());

accountsDto.setAccountType(accounts.getAccountType());

accountsDto.setBranchAddress(accounts.getBranchAddress());

return accountsDto;

}

public static Accounts mapToAccounts(AccountsDto accountsDto, Accounts accounts) {

accounts.setAccountNumber(accountsDto.getAccountNumber());

accounts.setAccountType(accountsDto.getAccountType());

accounts.setBranchAddress(accountsDto.getBranchAddress());

return accounts;

}

}  
   
package com.eazybytes.accounts.mapper;

public class CustomerMapper {

public static CustomerDto mapToCustomerDto(Customer customer, CustomerDto customerDto) {

customerDto.setName(customer.getName());

customerDto.setMobileNumber(customer.getMobileNumber());

customerDto.setEmail(customer.getEmail());

return customerDto;

}

public static Customer mapToCustomer(CustomerDto customerDto, Customer customer) {

customer.setName(customerDto.getName());

customer.setMobileNumber(customerDto.getMobileNumber());

customer.setEmail(customerDto.getEmail());

return customer;

}

}

There are libraries to do this mapping DTO to entity and entity to DTO automatically   
“modelmapper” and “mapStruct”

## Create API inside accounts microservice – Part 2

customerRepository.save(customer);  
The customer\_id will be generated automatically by the spring data jpa framework. To know the customer\_id we should save the returned object in a Customer object.  
save method is coming from JpaRepository. All the boiler plate code is taken care by the spring data jpa gframework.  
  
package com.eazybytes.accounts.service.impl;

@Service

@AllArgsConstructor

public class AccountsServiceImpl implements IAccountsService {

private AccountsRepository accountsRepository;

private CustomerRepository customerRepository;

@Override

public void createAccount(CustomerDto customerDto) {

Customer customer = CustomerMapper.mapToCustomer(customerDto, new Customer());

Customer savedCustomer = customerRepository.save(customer);

accountsRepository.save(createNewAccount(savedCustomer));

}

private Accounts createNewAccount(Customer customer) {

Accounts newAccount = new Accounts();

newAccount.setCustomerId(customer.getCustomerId());

long randomAccntNUmber = 1000000000L + new Random().nextInt(900000000);

newAccount.setAccountNumber(randomAccntNUmber);

newAccount.setAccountType(AccountsConstants.SAVINGS);

newAccount.setBranchAddress(AccountsConstants.ADDRESS);

return newAccount;

}

}

Create package for all exception handling classes com.eazybytes.accounts.exception  
public class CustomerAlreadyException extends RuntimeException {

….  
}  
Any exception that you create should extend RuntimeException.  
  
public CustomerAlreadyException(String message) {

super(message);

}  
Whenever anyone is trying to create an object of this Exception class with the message value the same message, we are passing to the Runtime Exception because we are extending this as a super class.

@ResponseStatus(value = HttpStatus.BAD\_REQUEST)  
This annotation is there because when ever this exception is thrown the client will get the message or response as BAD\_REQUEST.

package com.eazybytes.accounts.exception;

@ResponseStatus(value = HttpStatus.BAD\_REQUEST)

public class CustomerAlreadyException extends RuntimeException {

public CustomerAlreadyException(String message) {

super(message);

}

}

Now integrate this custom exception in the service class.  
But the duplicate customer cannot be found by the customer\_id since the NEW/FRESH incoming request does not have the customer\_id, we have identify the duplicate/existing customer using mobile number , so we need to find a customer with same mobile number if a customer exists with the incoming request mobile number it will treated as an existing customer.

package com.eazybytes.accounts.repository;

@Repository

public interface CustomerRepository extends JpaRepository<Customer, Long>{

Optional<Customer> findByMobileNumber(String mobileNumber);

}  
  
@Override

public void createAccount(CustomerDto customerDto) {

Customer customer = CustomerMapper.mapToCustomer(customerDto, new Customer());

Optional<Customer> optionalCustomer = customerRepository.findByMobileNumber(customerDto.getMobileNumber());

if(optionalCustomer.isPresent()) {

throw new CustomerAlreadyException("Customer already registered with given mobile number: "+ customerDto.getMobileNumber());

}

Customer savedCustomer = customerRepository.save(customer);

accountsRepository.save(createNewAccount(savedCustomer));

}

package com.eazybytes.accounts.service.impl;

@Service

@AllArgsConstructor

public class AccountsServiceImpl implements IAccountsService {

private AccountsRepository accountsRepository;

private CustomerRepository customerRepository;

@Override

public void createAccount(CustomerDto customerDto) {

Customer customer = CustomerMapper.mapToCustomer(customerDto, new Customer());

Optional<Customer> optionalCustomer = customerRepository.findByMobileNumber(customerDto.getMobileNumber());

if(optionalCustomer.isPresent()) {

throw new CustomerAlreadyException("Customer already registered with given mobile number: "+ customerDto.getMobileNumber());

}

Customer savedCustomer = customerRepository.save(customer);

accountsRepository.save(createNewAccount(savedCustomer));

}

private Accounts createNewAccount(Customer customer) {

Accounts newAccount = new Accounts();

newAccount.setCustomerId(customer.getCustomerId());

long randomAccntNUmber = 1000000000L + new Random().nextInt(900000000);

newAccount.setAccountNumber(randomAccntNUmber);

newAccount.setAccountType(AccountsConstants.SAVINGS);

newAccount.setBranchAddress(AccountsConstants.ADDRESS);

return newAccount;

}

}

But where is the logic to handle this exception, this method is called from the parent method which is in Controller.  
We can have a catch block and throw the expected message to the client but what if we need to throw and catch the same exception in some other controller or method, so it will be code duplication.  
  
To avoid this we need to write a Global Exception handling logic.  
  
package com.eazybytes.accounts.exception;

import org.springframework.web.bind.annotation.ControllerAdvice;

@ControllerAdvice

public class GlobalExceptionHandler {

}  
  
**@ControllerAdvice** – Using this annotation we are telling the spring boot framework whenever any exception happens in any of my controller in all such controllers, please invoke a method that I am going to write in this class.

But the question is how the spring boot framework will know that this method needs to be invoked when this exception is thrown.   
This is done by the annotation @ExceptionHandler(CustomerAlreadyException.class)

@ExceptionHandler(CustomerAlreadyException.class)

public ResponseEntity<ErrorResponseDto> handleCustomerAlreadyException(CustomerAlreadyException exception,

WebRequest webRequest)

{

….

}

package com.eazybytes.accounts.exception;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.context.request.WebRequest;

import com.eazybytes.accounts.dto.ErrorResponseDto;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(CustomerAlreadyException.class)

public ResponseEntity<ErrorResponseDto> handleCustomerAlreadyException(CustomerAlreadyException exception,

WebRequest webRequest)

{

ErrorResponseDto errorResponseDto = new ErrorResponseDto(

webRequest.getDescription(false),

HttpStatus.BAD\_REQUEST,

exception.getMessage(),

LocalDateTime.now());

return new ResponseEntity<>(errorResponseDto, HttpStatus.BAD\_REQUEST);

}

}  
  
  
Let’s fix and update the Controller class now.  
Autowire the IAccountsService we can do field auto wiring by mentioning @Autowired, but we can also do Constructor auto wiring by mentioning @AllArgsConstructor.  
Whenever we have a single constructor we do not need to mention @Autowired.  
  
package com.eazybytes.accounts.controller;

@RestController

@RequestMapping(path = "/api", produces = MediaType.APPLICATION\_JSON\_VALUE)

@AllArgsConstructor

public class AccountsController {

private IAccountsService iAccountsService;

@PostMapping("/create")

public ResponseEntity<ResponseDto> createAccount(@RequestBody CustomerDto customerDto)

{

iAccountsService.createAccount(customerDto);

return ResponseEntity

.status(HttpStatus.CREATED)

.body(new ResponseDto(AccountsConstants.MESSAGE\_201, AccountsConstants.MESSAGE\_201));

}

}

Update the createdAt and CreatedBy  
package com.eazybytes.accounts.service.impl;

@Service

@AllArgsConstructor

public class AccountsServiceImpl implements IAccountsService {

private AccountsRepository accountsRepository;

private CustomerRepository customerRepository;

@Override

public void createAccount(CustomerDto customerDto) {

Customer customer = CustomerMapper.mapToCustomer(customerDto, new Customer());

Optional<Customer> optionalCustomer = customerRepository.findByMobileNumber(customerDto.getMobileNumber());

if(optionalCustomer.isPresent()) {

throw new CustomerAlreadyException("Customer already registered with given mobile number: "+ customerDto.getMobileNumber());

}

customer.setCreatedAt(LocalDateTime.now());

customer.setCreatedBy("Anonymous");

Customer savedCustomer = customerRepository.save(customer);

accountsRepository.save(createNewAccount(savedCustomer));

}

private Accounts createNewAccount(Customer customer) {

Accounts newAccount = new Accounts();

newAccount.setCustomerId(customer.getCustomerId());

long randomAccntNUmber = 1000000000L + new Random().nextInt(900000000);

newAccount.setAccountNumber(randomAccntNUmber);

newAccount.setAccountType(AccountsConstants.SAVINGS);

newAccount.setBranchAddress(AccountsConstants.ADDRESS);

newAccount.setCreatedAt(LocalDateTime.now());

newAccount.setCreatedBy("Anonymous");

return newAccount;

}

}

POST http://localhost:8080/api/create  
  
{

    "name": "Madan Reddy",

    "email": "tutor@eazybytes",

    "mobileNumber": "9823150966"

}